VI.03.

И в этот раз - большею частью повторение. Но вбросим всё-таки ещё одно важнейшее свойство интерфейсов - композицию интерфейсов. На уровне качественного понимания, без крайних ситуаций, главное - чтобы было понятно, что это есть и зачем. Именно чисто качественно понятно.

Содержание занятия примерно такое:

- введём понятие композиции интерфейсов. Точнее говоря, расщепим совершенно естественным образом интерфейс Ordered с прошлого занятия на два интерфейса: один отвечает за сравнение ключей, а второй - за их визуализацию. Решение совершенно естественное, напрашивающееся и понятное.
- вспомним, в дополнение к SortedCollection и SortedList, ещё одну структуру, которая автоматом как-то перемещает свои элементы прямо при вставке, которая не позволяет ставить элементы в произвольном порядке - бинарную кучу.
- ну, и напомним (а это чистое повторение) бинарное дерево поиска с целью на следующем занятии рассказать про балансировку дерева поиска - про АВЛ-дерево.

И работу с бинарной кучей, и работу с двоичным деревом поиска оформим в виде соответствующих пакетов и добавим их внутрь `order'a`. Итого получим вот такую структуру каталогов:

```
order
│    order.go
├───binheap
│        binheap.go
├───bst
│        bst.go
├───sortable
│        sortable.go
└───sorted
        sorted.go
```

order.go

интерфейс Key = Ordered + Visual

Совершенно ясно, без всякой внутренней борьбы, что сравнение ключей и вывод ключей - это две совершенно независимые друг от друга субстанции, что выводим мы совсем не обязательно то, по чему сравниваем что-то, и уж точно совсем не всё, что мы выводим

нужно (и можно) сравнивать на больше/меньше. Ну так и разобьём эти два действия на два интерфейса: один обеспечивает сравнение, а другой - визуализацию, отвечает за внешний вид отображения ключа. На подробности, свойства и последствия применения композиции интерфейсов в этот раз не заморачиваемся, будет ещё время и место поговорить об этом отдельно и специально, а пока совсем по-простому, практически на уровне идеи, которая, оказывается, реализована в Go. А означает это следующее: при построении интерфейсов внутрь них можно встраивать (embedding) уже имеющиеся интерфейсы. И получается, что тип реализует какой-то интерфейс, если он реализует не только все методы этого интерфейса, но и все методы встроенных в него интерфейсов. Зачем это делать - более или менее понятно, понятно, что интерфейсы желательно делать помельче, если речь идёт о базовых интерфейсах, о тех, которые встраиваются в другие интерфейсы. Но в этот раз - ограничимся чисто содержательным моментом. И всё уже видно: дробить интерфейсы (осмысленно дробить, конечно) - это хорошо и логично. Чуть ниже, в разговоре о бинарной куче будет некий пассаж на эту тему, но можно его и здесь привести/рассмотреть.

```
package order

type Ordered interface {
    Before(b Ordered) bool
}

type Visual interface {
    Image() string
}

type Key interface {
    Ordered
    Visual
}

const (
    PreOrder         = iota // NLR
    InOrder                 // LNR = ascending order
    PostOrder               // LRN
    ReversePreOrder         // NRL
    ReverseInOrder          // RNL = descending order
    ReversePostOrder        // RLN
)

var ImageWidth int

func init() {
```

```
    ImageWidth = 5
}
```

Ну, и ещё кое-что по-мелочи добавлено: константы для определения порядка обхода бинарного дерева и переменная ImageWidth для указания ширины изображения при выводе ключей - это для визуализации дерева. Хорошо бы про неё сказать пару слов - тут и экспортируемая из пакета переменная (этакая “глобальная” переменная), тут и функция init().

Пакет sorted (файл sorted.go)

нас сегодня вообще не волнует. Хотя, раз уж в нём есть методы печати коллекции и списка, то заменим order.Ordered на order.Key.

binheap.go

Тут мы собираем всё, что связано с бинарной кучей. Всё это уже было в пятом семестре, да и вообще, сегодняшнее занятие во многом связано с повторением. Подробно про бинарную кучу говорили в 15-ом занятии пятого семестра. Описания, обоснования и прочие заметки про бинарную кучу выложены также и прямо здесь в md-формате и pdf-формате

Но есть один обещанный выше момент про композицию интерфейсов. Бинарная куча - это инструмент, вспомогательный инструмент для решения различных задач, в частности, задачи сортировки. А, поскольку это вспомогательный инструмент, то визуализация ему совсем ни к чему. Для функционирования бинарной кучи вполне достаточно только сравнивать элементы. Ну так и будем реализовывать бинарную кучу в слайсе `order.Ordered`‘ов. А вот, используя бинарную кучу для сортировки HeapSort, будем спокойно ассертить получаемые из бинарной кучи значения до необходимых типов. Так что работает наше расщепление интерфейсов, работает…

В приведённом коде строится max-куча (никакой родитель не идёт перед своими детьми в заданном порядке) и операции с ней:

```
package binheap

// Max-heap
import (
    "order"
)

type BinaryHeap []order.Ordered
```

```
func (b *BinaryHeap) Add(Element order.Ordered) {
    (*b) = append(*b, Element)
    (*b).pushUp(len(*b) - 1)
}

func (b BinaryHeap) Heapify() {
    for k := (len(b) / 2) - 1; k >= 0; k-- {
        b.pushDown(k)
    }
}

func (b BinaryHeap) GetMax() order.Ordered {
    if len(b) > 0 {
        return b[0]
    } else {
        var empty order.Ordered
        return empty
    }
}

func (b *BinaryHeap) ExtractMax() order.Ordered {
    if len(*b) > 0 {
        max := (*b)[0]
        (*b)[0] = (*b)[len(*b)-1]
        *b = (*b)[:len(*b)-1]
        b.pushDown(0)
        return max
    } else {
        var empty order.Ordered
        return empty
    }
}

func (b *BinaryHeap) Delete(place int) {
    if place >= len(*b) || place < 0 {
        return
    }
    x := (*b)[len(*b)-1]
    *b = (*b)[:len(*b)-1]
    (*b).Change(place, x)
}

func (b BinaryHeap) Change(place int, Key order.Ordered) {
    if place >= len(b) || place < 0 {
```

```
        return
    }
    b[place] = Key
    if place > 0 && b[(place-1)/2].Before(Key) {
        b.pushUp(place)
    } else {
        b.pushDown(place)
    }
}

func (b BinaryHeap) pushUp(place int) {
    if place >= len(b) || place <= 0 {
        return
    }
    x := b[place]
    parent := (place - 1) / 2
    for place > 0 && b[parent].Before(x) {
        b[place] = b[parent]
        place = parent
        parent = (place - 1) / 2
    }
    b[place] = x
}

func (b BinaryHeap) pushDown(place int) {
    if place >= len(b) || place < 0 {
        return
    }
    x := b[place]
    for {
        if 2*place+1 >= len(b) { // лист - сыновей нет
            break
        }
        maxson := 2*place + 1 // левый сын
        rson := maxson + 1
        if rson < len(b) && b[maxson].Before(b[rson]) { // правый сын больше лево
го
            maxson = rson
        }
        if !x.Before(b[maxson]) {
            break
        }
        b[place] = b[maxson]
        place = maxson
```

```
    }
    b[place] = x
}
```

Действие всех методов понятно из их дескрипторов.

sortable.go

И вот здесь мы обоим сортируемым структурам - коллекции (на слайсе) и связному списку `Key`'ев - добавим ещё одну сортировку. Heapsort, разумеется.

Sortable Collection. HeapSort.
Тут всё понятно и однозначно. Никаких вариантов не видно. Так что тут говорить особо не о чем, кроме, конечно, собственно о бинарной куче.

Вот код, который мы добавили в пакет `sortable`:

```
func (a SortableCollection) HeapSort() {
    var bheap binheap.BinaryHeap
    for _, x := range a {
        bheap.Add(x)
    }
    for k := len(a) - 1; k >= 0; k-- {
        a[k] = bheap.ExtractMax().(order.Key)
    }
}
```

Sortable List. HeapSort
Необходимые типы данных уже определены, интерфейсы реализованы:

```
type (
    Node struct {
        Value order.Key
        Next  SortableList
    }
    SortableList struct {
        First *Node
    }
)

func (a Node) Before(b order.Ordered) bool {
    return a.Value.Before(b.(Node).Value)
}
```

```
func (a Node) Image() string {
    return a.Value.Image()
}
```

Обращаем внимание на то, что узел (`Node`) реализует интерфейс `Key` . Странно? Ведь это значение (`Value`) есть `Key` . Да нет, не очень странно, работаем-то мы с узлами. Понятно, что сравниваем мы не узлы, а ключи, и визуализуем ключи, а не связи списка. Но так удобно думать, да и пмсать тоже получается код выглядит попроще, пояснее. Так что на этом моменте хорошо бы пусть немного, но приостановиться.

Специфика HeapSort'а такова, что ничего особо интересного списки в эту сортировку не привносят, даже с технической точки зрения. Ведь бинарная куча - это, по-любому, слайс, так что всяко его придётся создавать, дублируя тем самым элементы списка. Но дублирование элементов (клонирование, если хотите) - это нечто противоестественное. Ну так и и не станем клонировать узлы! Не зря же мы сделали узел (`Node`) `Ordered` 'ом (точнее `Key` ем, но `Key` включает в себя `Ordered`) - мы ставим узлы списка прямо целиком в бинарную кучу, а потом достаём их из неё. Мы не выделяем повторно память под узлы - гуляют в кучу и обратно исходные узлы. Мы сравниваем в куче узлы, а не значения, не ключи. А потом, не создавая новых узлов, аккуратненько связываем их в список в другом порядке. Момент любопытный.

Единственное замечание относится вот к этой строке:

```
    (*a).Add(bheap.ExtractMax().(Node))
```

Бинарная куча состоит из `Ordered` 'ов, соответственно, метод `bheap.ExtractMax()` возвращает `Ordered` , так что необходимо получаемый из него результат ассертить к типу узла списка.

Вот добавленный код:

```
func (a *SortableList) HeapSort() {
    var bheap binheap.BinaryHeap
    for !(*a).Empty() {
        bheap.Add(*(*a).First)
        *a = (*(*a).First).Next
    }
    *a = NewSortableList()
    for k := len(bheap) - 1; k >= 0; k-- {
        (*a).Add(bheap.ExtractMax().(Node))
    }
}
```

Бинарное дерево поиска. Объявление типов.

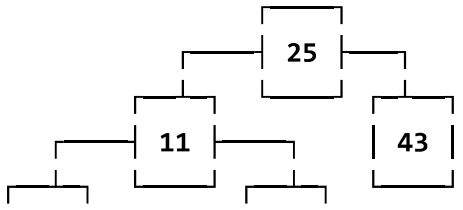
Тип узла дерева поиска и тип собственно дерева, не изменились, но, конечно значение ключа, хранящееся в узле дерева, теперь есть `Key` :

```
type (
    Node struct {
        Value order.Key
        Lson  Tree
        Rson  Tree
    }
    Tree struct {
        Root *Node
    }
)
```

Код всяких операций практически не меняется, только операции сравнения, разумеется, заменяются на обращения к методу Before интерфейса Key (даже только Ordered): вместо `if a < b` пишем `a.Before(b)` , вместо `if a <= b` пишем `!b.Before(a)` и т.д. А всё остальное вообще не меняется. Видимо нет даже смысла повторять здесь любые слова про устройство бинарного дерева поиска и операции с ним.
Тем не менее подробный материал по бинарное дерево поиска лежит прямо здесь в md-формате и pdf-формате
Код бинарного дерева поиска, основанного на интерфейсах `Key` и `Ordered` имеется в материалах к этому занятию - есть смысл по нему быстренько (именно быстренько, не задерживаясь вообще нигде) пробежаться. Ну, можно задержаться на том, как устроены нетривиальные операции удаления элемента и вывода (визуализации) дерева, но тоже особо не задерживаясь.

Задание на дерево поиска для самостоятельной работы

1. Модифицировать функцию печати дерева так, чтобы значения ключей печатались бы в рамке, что-то вроде такого:



2. Модифицировать метод Image() так, чтобы значение ключа печаталось бы не в одну строку, а в несколько, т.е. кроме параметра keyWidth задавался бы параметр keyHeight - сколько строк занимает изображение ключа. И метод Image возвращал бы строку, объединяющую в себе keyHeight строк, разделённых символами перевода строки.

Заключительные замечания

Всё срослось. Красиво? Ну, минимум довольно естественно получилось. И интерфейсы как инструмент реализации полиморфизма (можно уже это слово употреблять смело) работает здесь прекрасно. Но! Есть но. Всё это изрядно противоречит стратегии, можно даже сказать, идеологии Go, с его подчёркнутым отказом от наследования (и, естественно, от идеи объекта и ООП вообще). Как такие вопросы решать присущим именно Go, идиоматичным ему образом мы начнём обсуждать очень скоро, не на следующем занятии - следующее занятие проведём ещё в такой же манере, наберём чуть побольше содержательного материала, поговорим о сбалансированных деревьях поиска, а именно, тщательно разберёмся с АВЛ-деревьями. А вот дальше у нас появятся проблемы, которые в Go надо решать как-то иначе. И вот тут-то мы и посмотрим, что же именно предлагается для решения проблем, которые в ООП решаются с помощью наследования, а в Go совсем иначе. Так что в этом смысле Go не есть просто необъектный язык, Go - это пост-объектный язык программирования. И этот совсем иной подход дейтвительно выглядит очень здраво и естественно, позволяя, в конечном итоге, очень хорошо и довольно просто структурировать логику конструирования данных и методов их обработки. Какова именно эта выигрышная стратегия в Go - это и есть главный вопрос этого семестра. Именно им мы и займёмся уже совсем скоро.